Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$

Model: $h_\theta(x) = \theta_0 + \theta_1 x$
Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$

Model: $h_\theta(x) = \theta_0 + \theta_1 x$
Linear Regression

Parameters: \( \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \)  
Model: \( h_\theta(x) = \theta_0 + \theta_1 x \)
Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$

Model: $h_\theta(x) = \theta_0 + \theta_1 x$

Data:

<table>
<thead>
<tr>
<th>Population</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
</tr>
</tbody>
</table>

$y = -6.00 + 2.00x$
Linear Regression

Parameters: \( \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \)

Model: \( h_\theta(x) = \theta_0 + \theta_1 x \)

\[ y = -2.50 + 1.00x \]

Data

Population

Profit

\[ y = -2.50 + 1.00x \]

Data
Linear Regression

Parameters: \( \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \)  
Model: \( h_\theta(x) = \theta_0 + \theta_1 x \)

\[ y = -3.90 + 1.19x \]

Data

Profit vs. Population
The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \rightarrow \mathbb{R}$$

$$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$$

Mean Square Error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x(i)) - y(i))^2$$

where $m$ is the number of data points in the training set.
The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

  $$J : \mathbb{R}^2 \to \mathbb{R}$$

  $$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$$

- Mean Square Error:

  $$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_\theta(x^{(i)}) - y^{(i)} \right)^2$$
The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

  $$J : \mathbb{R}^2 \to \mathbb{R}$$

  $$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$$

- Mean Square Error:

  $$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_{\theta}(x^{(i)}) - y^{(i)} \right)^2$$

  $$= \frac{1}{2m} \sum_{i=1}^{m} \left( \theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$
The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

  $$J : \mathbb{R}^2 \rightarrow \mathbb{R}$$
  $$\hat{\theta} = \text{arg min}_{\theta \in \mathbb{R}^2} J(\theta)$$

- Mean Square Error:

  $$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

  $$= \frac{1}{2m} \sum_{i=1}^{m} \left( \theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$

  where $m$ is the number of data points in the training set.
The cost (or loss) function

\[ y = -5.00 + 1.50x \]

\[ J(\begin{bmatrix} -5.00 \\ 1.50 \end{bmatrix}) = 6.1561 \]
The cost (or loss) function

\[ y = -6.00 + 2.00x \]

\[ J\left( \begin{bmatrix} -6.00 \\ 2.00 \end{bmatrix} \right) = 19.3401 \]
The cost (or loss) function

\[ y = -2.50 + 1.00x \]

\[ J \left( \begin{bmatrix} -2.50 \\ 1.00 \end{bmatrix} \right) = 4.7692 \]
The cost (or loss) function

\[ y = -3.90 + 1.19x \]

\[ J\left( \begin{bmatrix} -3.90 \\ 1.19 \end{bmatrix} \right) = 4.4775 \]
The cost (or loss) function

So, how do we find $\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$ computationally?
The cost (or loss) function

So, how do we find \( \hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta) \) computationally?
\( \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \) for each \( j \)
(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

Step 0, $\alpha = 0.01$
\( \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \) for each \( j \)

Step 1, \( \alpha = 0.01 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad \text{for each } j \]

Step 20, \( \alpha = 0.01 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 200, \( \alpha = 0.01 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \] for each \( j \)

Step 10000, \( \alpha = 0.01 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10000, \( \alpha = 0.005 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10000, \( \alpha = 0.02 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10, \( \alpha = 0.025 \)
Backpropagation

How do we calculate \( \frac{\partial}{\partial \theta_j} J(\theta) \)?

In other words:
how sensitive is the loss function to the change of a parameter \( \theta_j \)?

**why backpropagation?**
we could do this by hand for linear regression...
but what about complex functions?
→ *propagate error backward*
(special case of *automatic differentiation*)
Backpropagation

\[
\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5
\]

Next, let's use dynamic programming to avoid re-computing intermediate results...

Christopher Olah
http://colah.github.io/posts/2015-08-Backprop/
Backpropagation

Applying chain rule:

\[
\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5
\]

Next, let's use dynamic programming to avoid re-computing intermediate results...
Backpropagation

\[
\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b}
\]

\[
e = c \cdot d
e = 6
\]

\[
c = a + b
c = 3
\]

\[
\frac{\partial c}{\partial a} = 1
\]

\[
\frac{\partial c}{\partial b} = 1
\]

\[
d = b + 1
d = 2
\]

\[
\frac{\partial d}{\partial b} = 1
\]

\[
a = 2
\]

\[
b = 1
\]

next, let's use dynamic programming to avoid re-computing intermediate results...
applying chain rule:

\[
\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5
\]

next, let's use *dynamic programming* to avoid re-computing intermediate results...
Backpropagation

forward-mode differentiation lets us compute partial derivatives $\frac{\partial x}{\partial b}$ for all nodes $x$

→ still inefficient if you have many inputs

Christopher Olah http://colah.github.io/posts/2015-08-Backprop/
backward-mode differentiation lets us efficiently compute $\frac{\partial e}{\partial x}$ for all inputs $x$ in one pass
→ also known as *error backpropagation*
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model; (here: a linear model)
- a suitable cost (or loss) function; (here: mean square error)
- an optimization algorithm; (here: a variant of SGD)
- the gradient(s) of the cost function (if required by the optimization algorithm).
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
- an optimization algorithm;
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
- an optimization algorithm;
- the gradient(s) of the cost function (if required by the optimization algorithm).
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model; \((\text{here: a linear model})\)
- a suitable cost (or loss) function; \((\text{here: mean square error})\)
- an optimization algorithm; \((\text{here: a variant of SGD})\)
- the gradient(s) of the cost function (if required by the optimization algorithm).
What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, ...)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices

Why is this useful?

- very expressive
- can represent (e.g.) parameterised probability distributions
- evaluation and parameter estimation can be built up from components

Relationship to linear regression

- more complex architectures with hidden units (neither input nor output)
- neural networks typically use non-linear activation functions
What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, …)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices

- Why is this useful?
  - very expressive
  - can represent (e.g.) parameterised probability distributions
  - evaluation and parameter estimation can be built up from components
What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, ...)  
  - maps vectors/matrices to vectors/matrices  
  - is parameterised by vectors/matrices

- Why is this useful?
  - very expressive
  - can represent (e.g.) parameterised probability distributions
  - evaluation and parameter estimation can be built up from components

**relationship to linear regression**

- more complex architectures with *hidden* units  
  (neither input nor output)
- neural networks typically use non-linear activation functions
An Artificial Neuron

- \( x \) is a vector input, \( y \) is a scalar output
- \( w \) and \( b \) are the parameters (\( b \) is a bias term)
- \( g \) is a (non-linear) activation function
Functions like XOR cannot be separated by a *linear* function.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)
Functions like XOR cannot be separated by a *linear* function

(neurons arranged in layers, and fire if input is $\geq 1$)
Why Non-linearity?

Functions like XOR cannot be separated by a *linear* function

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)
Why Non-linearity?

Functions like XOR cannot be separated by a *linear* function

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)
Activation functions

- desirable:
  - differentiable (for gradient-based training)
  - monotonic (for better training stability)
  - non-linear (for better expressivity)
More Complex Architectures

### Convolutional

- Fully connected layer
- K-Max pooling ($k=3$)
- Folding
- Wide convolution ($m=2$)
- Dynamic $k$-max pooling ($k=f(s)=5$)
- Wide convolution ($m=3$)
- Projected sentence matrix ($s=7$)

### Recurrent

- Target chars: 
  - "e": 1.0 2.2 -3.0 -4.1
  - "i": 0.5 0.3 -1.0 1.2
  - "l": 0.1 0.5 1.9 -1.1
  - "o": 0.2 -1.5 -0.1 2.2

- Output layer
- Hidden layer
- Input layer

- Input chars: 
  - "h": 1 0 0
  - "e": 0 1 0
  - "l": 0 0 1
  - "l": 0 0 0

Andrey Karpathy

[http://karpathy.github.io/2015/05/21/rnn-effectiveness/](http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

[Kalchbrenner et al., 2014]
Training of Neural Networks

- Parameter estimation
  - Use *gradient descent*
  - Requires *labelled training data* . . .
    . . . and *differentiable objective function*
- Network structure enables efficient computation
  - *Forward pass* to compute network output
  - *Backpropagation*, i.e. backward pass using chain rule, to calculate gradient
- Normally train *stochastically* using *mini-batches*
Practical Considerations

- hyperparameters:
  - number and size of layers
  - minibatch size
  - learning rate
  - ...

- initialisation of weight matrices
- stopping criterion
- regularization (dropout)
- bias units (always-on input)
# Toolkits for Neural Networks

## What does a Toolkit Provide
- Multi-dimensional matrices (tensors)
- Automatic differentiation
- Efficient GPU routines for tensor operations

<table>
<thead>
<tr>
<th>Toolkit</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Torch</td>
<td><a href="http://torch.ch/">http://torch.ch/</a></td>
</tr>
<tr>
<td>TensorFlow</td>
<td><a href="https://www.tensorflow.org/">https://www.tensorflow.org/</a></td>
</tr>
<tr>
<td>Theano</td>
<td><a href="http://deeplearning.net/software/theano/">http://deeplearning.net/software/theano/</a></td>
</tr>
</tbody>
</table>

There are many more!

further reading on backpropagation:
http://colah.github.io/posts/2015-08-Backprop/
some slides borrowed from:

- Sennrich, Birch, and Junczys-Dowmunt (2016): Advances in Neural Machine Translation
- Sennrich and Haddow (2017): Practical Neural Machine Translation